

Support de cours

Cours:

Introduction à la programmation orientée objet (en C++)

Vidéo:

W11-04-resume-CPP-pt2

Concepts (extraits des sous-titres générés automatiquement) :

Classe rectangle. Client de la classe rectangle. Nom local. Première chose. Programmeur concepteur de la classe rectangle. Partie de l'externalisation de la définition. Programme principal. Côté du programmeur utilisateur. Nouveau type. Mécanisme de surcharge. Variable de type rectangle. Confusion possible. Intérêt d'externaliser. Convention de nommage particulière. Exemple d'une méthode.



[vers la recherche de séquences vidéo](#)

(dans Introduction à la programmation orientée objet (en C++).)



[vers la vidéo](#)

Center for Digital Education. Plus de matériel de soutien pédagogique ici :

<https://www.epfl.ch/education/educational-initiatives/cede/educational-technologies-gallery/boocs-en/>

Encapsulation et abstraction : résumé

(Partie 2)

Introduction à la programmation orientée objet (en C++)

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s

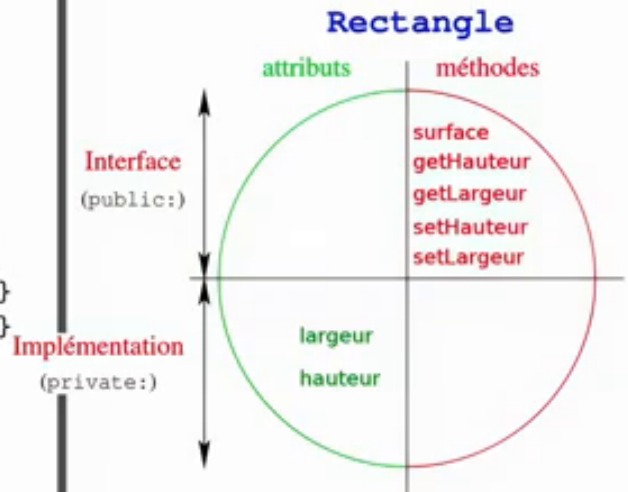


Un exemple complet de classe (1/2)

```
#include <iostream>
using namespace std;

// définition de la classe
class Rectangle {
public:
    // définition des méthodes
    double surface() const
    { return hauteur * largeur; }
    double getHauteur() const { return hauteur; }
    double getLargeur() const { return largeur; }
    void setHauteur(double h) { hauteur = h; }
    void setLargeur(double l) { largeur = l; }

private:
    // déclaration des attributs
    double hauteur;
    double largeur;
};
```



Le code que vous avez sous les yeux,

notes

résumé

0m 1s



Un exemple complet de classe (2/2)

//utilisation de la classe

```
int main()
{
    Rectangle rect;
    double lu;
    cout << "Quelle hauteur? "; cin >> lu;
    rect.setHauteur(lu);
    cout << "Quelle largeur? "; cin >> lu;
    rect.setLargeur(lu);

    cout << "surface = " << rect.surface()
         << endl;

    return 0;
}
```

objet, instance



5.0 → lu

notes

résumé

0m 5s



Un exemple complet de classe (2/2)

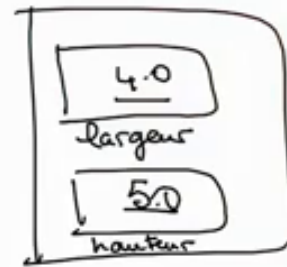
//utilisation de la classe

```
int main()
{
    Rectangle rect;
    double lu;
    cout << "Quelle hauteur? "; cin >> lu;
    rect.setHauteur(lu);
    cout << "Quelle largeur? "; cin >> lu;
    rect.setLargeur(lu);

    cout << "surface = " << rect.surface()
         << endl;

    return 0;
}
```

objet, instance



5.0 → lu

Et pareil ici, pour la largeur. Par exemple ici. Une fois que le rectangle est proprement initialisé,

notes

résumé

Exemple avec définitions externes à la classe

```
class Rectangle
{
public:
    // prototypes des méthodes
    double surface() const;

    // accesseurs
    double hauteur() const;
    double largeur() const;

    // manipulateurs
    void hauteur(double);
    void largeur(double);

private:
    // déclaration des attributs
    double hauteur_;
    double largeur_;
};
```

const << rect.hauteur() << endl;

```
double Rectangle::surface() const
{
    return hauteur_ * largeur_;
}

double Rectangle::hauteur() const
{
    return hauteur_;
}

... // idem pour largeur

void Rectangle::hauteur(double h)
{
    hauteur_ = h;
}

... // idem pour largeur
```

il est possible d'invoquer d'autres fonctionnalités sur ce rectangle, comme typiquement ici, le calcul de la surface, et on passe toujours par la fameuse notation pointée qui signifie ici, je calcule la surface du rectangle. Et voici pour finir, une écriture affinée de la classe Rectangle, qui tire partie de l'externalisation de la définition des méthodes. Donc ici les méthodes sont uniquement prototypées dans la classe rectangle, leur définition est placée à l'extérieur de la classe Rectangle. Le lien entre la méthode et la classe à laquelle elle appartient, est fait au moyen de l'opérateur de résolution de portée. L'opérateur de résolution de portée (::), permet donc ici de lier un nom local, qui est le nom par exemple d'une méthode appartenant à la classe, avec un nom plus global, qui est celui de la classe à laquelle cette méthode appartient. L'intérêt d'externaliser, est que la déclaration de la classe est désormais épurée de tous les détails de la définition des méthodes. Donc cette façon de déclarer la classe en y mettant uniquement le prototype des méthodes, et bien évidemment, les attributs, va constituer ce que l'on appelle, le prototype de la classe. Les définitions des méthodes, l'ensemble des définitions des méthodes de la classe, vont constituer la définition de la classe. Donc exactement pareil que ce que l'on a vu avec les fonctions, le prototype est ce qui va permettre l'accord entre le programmeur utilisateur et le programmeur concepteur de la classe Rectangle. Les définitions ne sont connues que du programmeur concepteur. Celui qui utilise la classe Rectangle n'a besoin de connaître que la partie publique de son prototype. Notez que dans une bonne conception, on prendra soin de marquer comme const, tout ce qui n'est pas un manipulateur, tout ce qui n'est pas amené à modifier la valeur des attributs. Plusieurs petits détails peuvent être remarqués ici, notamment

notes

résumé

2m 13s



Exemple avec définitions externes à la classe

```
class Rectangle
{
public:
    // prototypes des méthodes
    double surface() const;

    // accesseurs
    double hauteur() const;
    double largeur() const;

    // manipulateurs
    void hauteur(double);
    void largeur(double);

private:
    // déclaration des attributs
    double hauteur_;
    double largeur_;
};
```

const << rect.hauteur() << endl;

```
double Rectangle::surface() const
{
    return hauteur_ * largeur_;
}

double Rectangle::hauteur() const
{
    return hauteur_;
}

... // idem pour largeur

void Rectangle::hauteur(double h)
{
    hauteur_ = h;
}

... // idem pour largeur
```

le fait que certains programmeurs jugent utile d'adopter une convention de nommage particulière pour les attributs, comme le fait de les terminer par le caractère souligné. Donc c'est pour éviter d'éventuels masquages avec des noms d'arguments de méthode. Vous noterez également ici, la façon particulière qu'a le programmeur de nommer ses accesseurs et ses manipulateurs. Dans la version précédente du programme, la méthode qui permettait d'accéder à l'attribut hauteur du Rectangle, s'appelait getHauteur, ici elle s'appelle hauteur tout court. Pareil pour la largeur. De même, la méthode qui permettait de modifier la valeur de l'attribut hauteur, en y mettant une valeur particulière dedans, s'appelait setHauteur, ici elle s'appelle hauteur tout court. Certains programmeurs trouvent cette façon de nommer les accesseurs et les manipulateurs, plus claire, plus lisible, au niveau de l'utilisation. Par exemple, ici, si dans un programme qui utilise la classe Rectangle, je veux faire afficher la hauteur d'un rectangle donné, qui aurait été préalablement déclarée, donc ici, il me suffit d'écrire rect.hauteur(). Je n'ai pas besoin d'écrire rect.get.hauteur(),

notes

résumé

Exemple avec définitions externes à la classe

```
class Rectangle
{
public:
    // prototypes des méthodes
    double surface() const;

    // accesseurs
    double hauteur() const;
    double largeur() const;

    // manipulateurs
    void hauteur(double);
    void largeur(double);

private:
    // déclaration des attributs
    double hauteur_;
    double largeur_;
};
```

```
double Rectangle::surface() const
{
    return hauteur_ * largeur_;
}

double Rectangle::hauteur() const
{
    return hauteur_;
}

... // idem pour largeur

void Rectangle::hauteur(double h)
{
    hauteur_ = h;
}

... // idem pour largeur
```

ce qui est un peu plus lourd. De même si je veux modifier dans un programme qui utilise la notion de rectangle, la hauteur d'un rectangle donnée, j'écrirai tout simplement ceci, ce qui est jugé comme plus lisible et plus clair par certains programmeurs. On utilise bien évidemment ici le mécanisme de surcharge, le fait de donner le même nom à deux méthodes différentes, mécanisme de surcharge qui est possible pour les méthodes d'une classe, comme il l'est pour les fonctions usuelles. Donc il n'y a pas de confusion possible au niveau de l'appel. La méthode hauteur ici, qui est un manipulateur se distingue de la méthode hauteur, qui est un accesseur, par la liste des arguments, qui est différente dans les deux cas, mais également par la présence ou non du modificateur const. Donc pas de confusion possible au moment de l'appel à ces méthodes. au moment de l'appel à ces méthodes.

notes

résumé

5m 25s

